CAE Specification

Systems Management:
Distributed Software Administration

The Open Group

CAE Specification

Systems Management: Distributed Software Administration

ISBN: 1-85912-149-7

Document Number: C701

Published in the U.K. by The Open Group, RJanuary 1998.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

List of Tables

B.2 Software Structures

B.2.1 Classes and Attributes

An example of the structure of the software objects for this Software Administration specification is illustrated in Figure B-2.

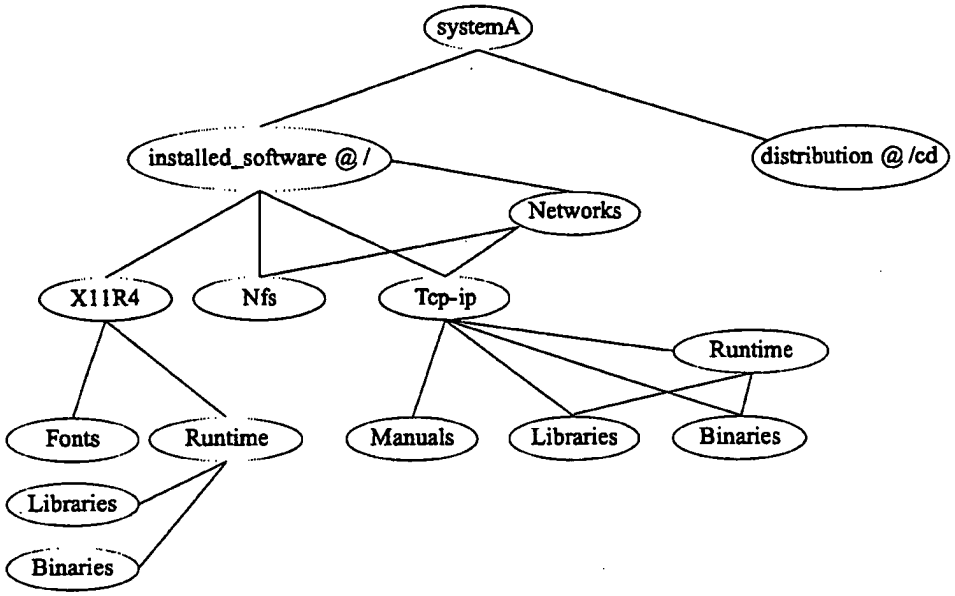

Figure B-2 Example of Software Structure

At the top of the hierarchy is a host, which is a system that conforms to this Software Administration specification. It is the starting point for finding all the software on that system that falls within this Software Administration specification. A host contains software_collections.

There are two distinct types of software_collections, as listed in the following, that may exist within a conformant system:

distribution
A distribution consists of software products, in a form ready for installation. A distribution may also contain software bundles. There may be many distributions within a host.

installed_software
An installed_software object consists of products installed from a distribution. An installed_software object may also contain software bundles. There may be other installed_software objects for use by this system or for other systems.

Software is organized into a hierarchy of objects, as described in the following, that are operated on by the utilities defined in this Software Administration specification:

product
A product consists of filesets and control scripts, plus all the associated metadata. The content of a product may be specified as a collection of subproducts, filesets, or a combination of the two.

bundle
A bundle is a grouping of other software objects and is a convenient way to reference a set of software.

fileset
A fileset consists of the actual files plus control scripts. Filesets are generally the lowest level of software object that can be operated on by the utilities.

subproduct
Subproducts are a grouping of other subproducts, or of filesets, or of some combination, that resolve to a group of filesets. Subproducts are a convenient way to aggregate filesets.

The software_files define the files and control_files that are contained in the software objects that are operated on during a software administration utility. There are two classes of software_files as described in the following:

control_file
Control_files consist of control scripts and other files that are used in various ways by the utilities. Control scripts are executed by the utilities at various points in a task. Control scripts provide a way to perform steps, in addition to those executed by the utilities, at various points in the task such as preinstall checking, postinstall customization, configuration, and verification. Either a single script with multiple entry points, or multiple scripts can be defined.

Most control scripts are run on the target, which may be a different architecture than the client on which the software operates. They should, therefore, use POSIX.2 utilities, except where they can determine that they are running on the client.

In addition to scripts, other control_files provide input to the control scripts, or to the utilities directly (for example, the `response` and `space` control_files).

file
Files are the lowest level of object defined by this Software Administration specification. Files contain the attributes describing the file including the contents of the file and its installed location.

The distributions and installed_software objects are the sources or targets of a software administration command. The software objects (products, filesets, bundles, and subproducts) are the objects that are being applied to those targets.

This Software Administration specification describes the structure and the attributes for software_collections, software objects, and software_files. It also describes the behaviors for the utilities that operate on these objects. However, these structure definitions are not managed object classes in the ISO sense because the behaviors are not described in terms of methods within object classes[17].

Figure B-3 on page 173 shows the components of the software object hierarchy. The containment arrows designate objects that are defined within the context of their containing objects. An object can only exist within one containing object. The identifier of an object (for

17. Object classes are templates for the creation of object instances. They are analogous to the definition statements used in programming languages to define data structures that will be created later. Objects contain more than data structures, in that they also possess methods (procedures that are executed by objects). A well-formed object class has methods defined that handle all object data manipulation, including creation, modification, and listing, so that the actual storage of the data is appropriately hidden from the application using the objects.

example, the *tag* attribute of a fileset) only needs to be unique within the scope of the containing object.

The reference arrows designate objects that are included when this object is operated on. An object may be referred to by more than one object. Bundles need not refer to entire products, but can refer to individual filesets or subproducts. Fileset and subproduct objects can be referenced directly by bundles by also identifying the product of which the fileset or subproduct is a part.

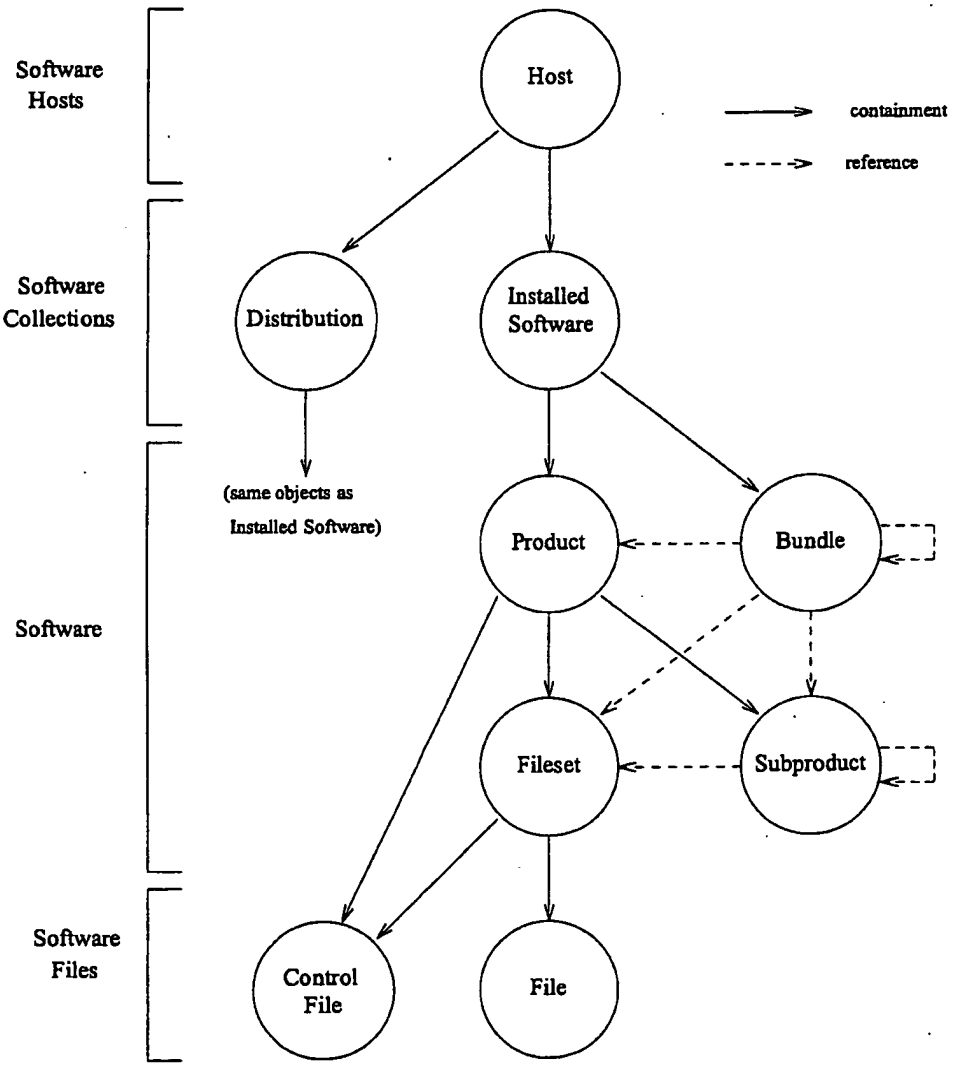

Figure B-3 Software Object Containment

Figure B-4 on page 174 shows the software administration common classes and the software objects that inherit attributes from these common classes.

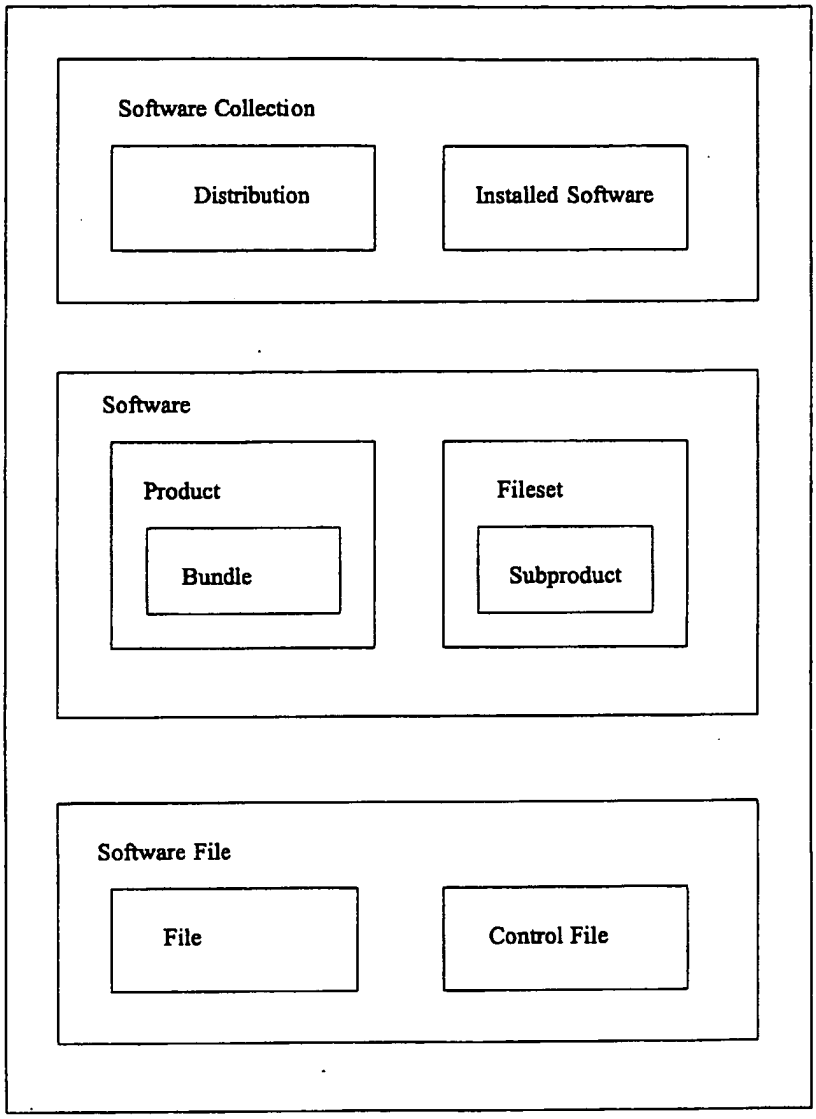

Figure B-4 Software Object Inheritance

Interoperability between implementations of this Software Administration specification may be achieved through the definition of methods for the first two of these common classes, "software_collections" and software. The software_collections are the source and target objects for software administration, while the software objects are the objects that are operated on within the context of the software_collections. Operations on individual software_files independent of operations on software objects is undefined.

This Software Administration specification also does not define how remote file systems are managed. In the simplest case, each file system is "local" to a single host, and all installations may be directed to the file system through an agent process on that host. Thus, files on a file system are contained within one of the installed software collections contained below that host.

On the other hand, an implementation may also choose to allow installation to a remote file system over a remote file system protocol. That is, the target process is running on a host that is different from the one that contains the file system. In this case, the files on that file system may be contained within the same software collection as before, or may be contained within a local software collection. In another implementation, all software collections may be stored within a global naming service instead of below any particular host.

An implementation may choose to define a software host object, or manage software as part of a more general host object. The attributes of a host object that are of interest to this Software Administration specification are shown in Table B-1.

Table B-1 Possible Attributes of a Host Class

Attribute	Length	Permitted Values	Default Value
host	Undefined	Portable character set	None
os_name	32	Portable character set	None
os_release	32	Portable character set	None
os_version	32	Portable character set	None
machine_type	32	Portable character set	None
distributions	Undefined	List of distribution directories	Empty list
installed_software	Undefined	List of installed_software directories and catalog identifiers	Empty list

The following are the attributes of the hosts that contain software_collections managed by this Software Administration specification:

distributions
The list of *distribution.path* attributes for distributions in the software host object.

These describe the PATHNAME portion of a software_collection source or target.

host
Identifier used to specify the host portion of a software source or target.

Identification of a remote host system is dependent on the networking services implementation and thus the syntax and semantics of the host name is undefined within this Software Administration specification.

installed_software
The list of *installed_software*path and *installed_software*catalog attributes for installed_software objects in the software host object.

These describe the PATHNAME portion of a software_collection target.

machine_type
Corresponds to the *machine* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the hardware type on which the system is running.

os_name
Corresponds to the *sysname* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the name of this implementation of the operating system.

os_release
Corresponds to the *release* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the release level of the operating system implementation.

os_version
Corresponds to the *version* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the version level of this release of the operating system.

B.2.2 Software_Collection

This class definition exists for convenience in defining the classes that inherit from it. It is not intended that any direct instances of this class be created, but only of the classes that inherit from it.

Multiple versions of products and bundles are possible when subsequent releases of a product or bundle have different revision numbers, and when products or bundles targeted for different machine types or other OS attributes define the architecture attribute differently.

The *layout_version* attribute is the version number of this Software Administration specification to which the distribution conforms. The name of this Software Administration specification (for example, P1387.2-19xx) was considered but there was concern that the delay between IEEE acceptance and ISO acceptance would make it hard to pick the year correctly. It is not clear when to change the number from 1.0 to 1.1 or even from 1.x to 2.0.

It is possible for an INDEX file describing a distribution to contain products with different values of *layout_version*. The software_collection *layout_version* refers only to the format of the distribution attributes and the **product** keyword. After the **product** keyword, the product *layout_version* defines the format of the definitions of all objects within that product.

B.2.3 Distribution

POSIX.1 allows for different pathname and filename sizes. Thus it is possible for a distribution to be created on one system and not be readable or installed on another system (each of which conforms with this Software Administration specification) because of differences in their POSIX.1 {NAME_MAX} and {PATH_MAX}. Consideration was given to attributes defining the longest sizes of file names and paths on a distribution, but these were not included since their use could neither ensure failure nor success of installing or copying a particular product from the distribution. Another issue implementors should consider is the maximum name and path that may be contained within a supported archive.

The need for the *media_sequence_number* attribute is to number the tapes (or disks or whatever) if a distribution is on more than one of them. If there is only one, then its number is 1.

The following attributes at one point were listed as distribution attributes. However, it was determined that the only time it could be guaranteed that these attributes were accurate was for an initial distribution definition. As soon as a *swcopy* or *swremove* operation occurred on a distribution, the attributes could be invalid because it would be impossible to modify these attributes in any logical manner based on the operation. It is recognized that these attributes are valuable and many vendors may choose to put them in as vendor extensions.

tag
A short name associated with the distribution, used for selecting the distribution from the command line

title
A longer name used for display purposes.

description
A more detailed description of the contents of the distribution.

revision
A revision associated with the distribution.

media_type
Describes the type of media being used (for example, CD-ROM, 8 mm, etc.)

copyright
The copyright notice for the distribution.

create_time
The date, in seconds since the Epoch, when the distribution was made.

number
The vendor part number for the distribution.

architecture
A sequence of characters used by a vendor to describe the machine or product. This is presumably more "user friendly" than the values returned by the *uname* utility.

Usually distributions will be created upon creation of the first product with *swpackage* or *swcopy*. Usually distributions will be removed as a part of removing the last product with *swremove*. An implementation may choose to provide more explicit control for creation and deletion of empty distributions. The *swcopy* and *swremove* utilities should be used for this purpose. The *swmodify* utility may also be used.

B.2.4 Media

This section is inserted to maintain parallel numbering with the main section numbering in Chapter 2 on page 5. No additional rationale is required under this heading.

B.2.5 Installed_Software

The installed_software catalog may be located by something as simple as a pathname where the catalog is stored as a file, or it could be located in a more complicated fashion such as with a key from a directory service used to identify all or part of a database.

B.2.6 Vendor

The *vendor.tag* attribute is intended to distinguish software objects from different vendors that happen to have the same *product.tag*. A vendor should attempt to choose a *vendor.tag* that is unique among all vendors.

B.2.7 Category

This section is inserted to maintain parallel numbering with the main section numbering in Chapter 2 on page 5. No additional rationale is required under this heading.

B.2.8 Software

This class definition exists for convenience in defining the classes that inherit from it. It is not intended that any direct instances of this class be created, but only of the classes that inherit from it.

This standard has defined four related software objects:

- Products
- Filesets
- Bundles
- Subproducts

See Figure B-4 on page 174. Implementations are encouraged to present these to the user as hierarchy of similar "software" objects, and to actually implement these so that they differ only as needed. That is to say, an implementation should use inheritance from a common class as much as possible. The rationale for the four differently named software objects is as follows:

- Products and filesets are concepts firmly entrenched in existing practice. All of the many practices that have contributed to this standard have included these two levels. Manageable software objects necessarily includes some files to manage. This is the basis of a software product. Additionally, most application software has both required and optional pieces, so often only a subset of the product may be installed. Thus, a fileset is chosen as a "set of files" and a product is a collection of filesets that have a number of shared attributes, and are distributed in a single distribution (usually from a single vendor).
- It was agreed that a "recursive notational convenience" was very desirable. Additionally, many (but not all) existing practices had realized the need for various overlapping groupings of software into new "configurations." Bundles and subproducts are merely "macro" or "recursive" products and filesets, respectively. Just as products and filesets are a bit different, the use of bundles and subproducts are a bit different. Bundles provide a way to make products out of existing products or parts of products. Subproducts provide a way to provide selectable units that may overlap in fileset contents. For example, a fileset may be part of "runtime" support as well as "development" environment subproducts. Finally, bundles and subproducts are recursive in that they may contain other bundles and subproducts, respectively.
- The containment of filesets and subproducts within products allows for derived naming of components of a product that is, a simple *tag* for a component relative to a more complex name (*tag*, *revision*, *vendor_tag*, *architecture*) for a product. In addition, this leads to distributions with a simple directory structure for filesets within products.

The need to localize the following descriptive software and vendor attributes was recognized *title*, *description* and *copyright*. However, since the existing practice for localization of software information files in portable media is immature, this has been deferred to a possible future revision of this Software Administration specification.

Until a future revision of this Software Administration specification addresses localization, one recommended way to internationalize these attributes is to create vendor-defined attributes with the format:

```
keyword.<LANG>
```

where keyword is "description.", "title", or "copyright", and <LANG> is the value of the LANG environment variable. An implementation should then recognize if LANG is set to a value other than its default and search for a corresponding attribute. If that attribute does not exist, then the default one will be used. For example:

```
product
    tag GreatProduct
    title "This is great!"
    title.FRENCH "C'est magnifique!"
    title.GERMAN "Sehr gut!"
    description"Long boring paragraph why this is great"
    description.FRENCH "...
    description.GERMAN "...
    . . .
```

Note that the *tag*, *revision*, and other attributes that affect the defined behavior of the implementation, shall not be internationalized. For this revision of this Software Administration specification, this includes all defined attributes except *title*, *description*, and *copyright*.

The size for the software may be larger than that supported by the POSIX.1 *size_t* structure since software can contain many files. It is recommended that an implementation allocate at least 64 b for the internal storage of the software *size* attribute.

B.2.9 Products

The value of the *revision* attribute is interpreted as a . (period) separated string, as defined in Section 2.9.0 on page 17, and further in Section 3.4.1 on page 38. This definition permits the use of such a string, but does not require it. The string can be constructed entirely without the use of periods. An example of the comparison is:

```
A1.003.01 < A.004.00 < B.000.00
A1_003_01 < A_004_00 < B_000_00
First < Second < Third
First < Fourth < Second
```

Historically, some implementations computed the value of *instance_id* sequentially, while other implementations have used an algorithm based on the product *tag*, *vendor_tag*, and the various machine type attributes. No implementation is specified, other than to guarantee that the *tag* and *instance_id* uniquely identify the product within the distribution or installed_software object. This is to make it easier to specify a particular product when there are other products sharing the same *tag* as would be the case when there are different product instances in a distribution for several machine types or multiple concurrent versions on a host.

The *vendor_tag* attribute is intended to be universally unique to distinguish product and bundle software objects that otherwise would be treated as the same object if the *tag*, *revision*, and *architecture* attributes were the same. Guaranteeing universal uniqueness is difficult at best, and no need was seen at present to cause the value of *vendor_tag* to be either some sort of machine-generated universally unique value or officially registered.

Multiple versions of the "same" product or bundle (ones with the same value for the *tag* attribute) is supported by each version possessing values of the version distinguishing attributes

unique within that installed software catalog.

The *architecture* attribute should include information related to four *uname*() structure members. The *architecture* attribute is needed for `software_specs` since the patterns used for determining compatibility in the attributes related to *uname*() can be somewhat complex and contain patterns, while `software_specs` themselves can contain patterns.

It is recommended that a set of guidelines be used for the architecture attributes to maintain a consistent "syntax" for related architectures. This increases the usability of this field for users selecting software. An example guideline is to order any information contained in the value of the attribute in a consistent way, separated by a consistent delimiter. For example:

```
architecture sunos_4.1_sun4
```

for a product with the attributes:

```
os_name sunos
os_rev 4.1.*
os_ver *
machine_type sun4*
```

Another example is:

```
architecture hp-ux_9_pa-risc
```

for a product with the attributes:

```
os_name hp-ux
os_rev 9.*|10.*
os_ver [a..e]
machine_type 9000/[6..8]???
```

Product machine attributes describe the target systems on which this product may be installed. Each of these keywords are related to a POSIX.1 *uname*() member and may be defined as a simple string, or a software pattern matching notation. How compatible software is determined depends on whether the products are being installed on the system that will be using them, or whether the installation will be used by other systems with perhaps different attributes.

If a uname attribute is undefined, the behavior is essentially the same as if it were defined to be * (meaning compatible with all systems).

The product directory for an application should be the directory that is part of all paths in the product. Thus, if an application has three filesets that contain files below `/appl/console`, `/appl/agent`, and `/appl/data` respectively, the *product.directory* attribute should be set to `/appl`. If a user relocates the product with a command like:

```
swinstall appl,r=1.0,l=/disk2/appl
```

then all three filesets have the same location attribute. If the user relocates the product to three different locations:

```
swinstall appl.console,r=1.0,l=/disk1/appl
swinstall appl.agent,r=1.0,l=/disk2/appl
swinstall appl.data,r=1.0,l=/disk3/appl
```

then each fileset will have a different location attribute. There will be three product instances containing the three filesets (since products versions are distinguished by location), but the user can still identify all three filesets as one with the specification:

```
swverify appl,r=1.0,l=*
```

Alternatively, the user could relate all these locations with the same version qualifier, such as "q=current" as follows:

```
swinstall appl.console,r=1.0,l=/disk1/appl,q=current
swinstall appl.agent,r=1.0,l=/disk2/appl,q=current
swinstall appl.data,r=1.0,l=/disk3/appl,q=current
```

and subsequently identify all pieces with:

```
swverify appl,q=current
```

The *postkernel* attribute supports the ability to install one operating system in proxy (to an alternate root) by another implementation that does not understand that operating system. All products that contain kernel filesets that will be installed into the same installed_software object should have the same path defined. There should be one core OS kernel fileset that includes this path in its set of files so that it has been installed by the time the *postkernel* script is executed.

In general, a product with no `preinstall` or `postinstall` scripts is recoverable. However, if there are `preinstall` or `postinstall` scripts, then `unpreinstall` and `unpostinstall` scripts shall be provided if any steps need to be undone to support autorecovery.

There was an issue whether dependencies should be an attribute of a product. The following types of dependencies have been discussed:

- Fileset to fileset within a product.
- Fileset to (some other) product.
- Fileset in one product to fileset in another product.
- Product to product.
- Product to fileset in some other product.
- Product to fileset in that product (essentially mandatory fileset).

The last three dependency types are not necessary if the first three types exist (which they do), since those dependencies can be specified in terms of the others. For example, if an entire product depends on a second product, then the second product can be defined as a dependency for all filesets in the first product.

The developers of this Software Administration specification recognized that numerous additional dependency requirements are possible, particularly for software updates. These may be handled via `checkinstall` scripts, and can be considered for future revisions of this Software Administration specification.

The intention behind the inclusion of the *layout_version* attribute within a product is that it be required if its value is different than that for its associated software_collection .

B.2.10 Bundles

Bundles serve two purposes they allow the software supplier to group different subsets of products into new configurations or products, and they allow the software administrator to build useful groups of software (configurations) from already defined bundles and products.

The bundle class does not have *location* or *directory* attributes. This is because `software_specs` within the bundles can refer to products with different default directory attributes or even products that have been relocated.

Bundles have "uname" attributes that only have any value if the bundle aggregate has a different compatibility than that of any of its contents. Besides offering more control to the person defining the bundle, it is useful in a GUI that wants to only display compatible software by default. For example, a bundle may contain one product that operates on a system with an uname attribute of "A" and another product that operates on systems with uname attributes of "A" or "B". In this case, it might be useful to define the bundle attribute to be "A". Since it is possible that not all the bundles contents exist in a particular distribution or installed_software object, it may not be possible to determine the compatibility of the bundle in all cases unless the bundle attributes are also defined.

The *vendor_tag* attribute is intended to be universally unique to prevent naming clashes for similarly named products and bundles from different vendors. Guaranteeing universal uniqueness is difficult at best; it was deemed unnecessary at present to cause the value of *vendor_tag* to be either some sort of machine-generated universally unique value or officially registered.

The intention behind the inclusion of the *layout_version* attribute within a bundle is that it be required if its value is different than that for its associated software_collection .

The value of the *bundle*contents attribute is not modified when a location is specified for a bundle, allowing future resolutions of its contents to remain consistent. For example, assume bundles `"CAT"` and `"DOG"`, and products `"FOO"` and `"BAR"`, all with directory attributes defined as `"/"`:

```
bundle
    tag CAT
    contents DOG,l=/dog BAR,l=/bar
bundle
    tag DOG
    contents FOO,l=/foo
```

When the bundle `"CAT"` is installed and relocated to `/cat`, the following objects are installed:

```
CAT,l=/cat
DOG,l=/cat/dog
FOO,l=/cat/dog/foo
BAR,l=/cat/bar
```

So, when resolving `"CAT,l=/cat"` in installed software, applying the proper locations to the `software_specs` in the contents will result in the same `software_specs` in the installed software.

Bundle definitions are only copied or installed when explicitly specified since they are external to the product and not always applicable to the use of the product installed. The creator of a product has no control over what bundles reference it. For example, a product may be a member of numerous bundles, and many of those bundles will likely have nothing to do with the bundles and products chosen to be installed. Also, see Section B.2.12 on page 183.

Bundles and subproducts have lists defining their contents that are always copied (*contents* is a static attribute). So, if a partial bundle or product is copied, the value of the *contents* attribute does not change. However, by comparing that attribute to what objects are actually installed, "completeness" of a bundle or subproduct can be determined.